基数估计算法概览

作者 张洋 | 发布于 2012-11-23

基数估计 翻译

翻译自《Damn Cool Algorithms: Cardinality Estimation》，原文链接：http://blog.notdot.net/2012/09/Dam-Cool-Algorithms-Cardinality-Estimation

假如你有一个巨大的含有重复数据项数据集，这个数据集过于庞大以至于无法全部放到内存中处理。现在你想知道这个数据集里有多少不同的元素，但是数据集没有排好序，而且对如此大的一个数据集进行排序和计数几乎是不可行的。你要如何估计数据集中有多少不同的数据项？很多应用场景都涉及这个问题，例如设计数据库的查询策略：一个良好的数据库查询策略不但和总的数据量有关，同时也依赖于数据中不同数据项的数量。

我建议在继续阅读本文前你可以稍微是思考一下这个问题，因为接下来我们要谈的算法相当有创意，而且实在是不怎么直观。

一个简单直观的基数估计方法

让我们从一个简单直观的例子开始吧。假设你通过如下步骤生成了一个数据集：

1、随机生成n个服从均匀分布的数字

2、随便重复其中一些数字，重复的数字和重复次数都不确定

3、打乱这些数字的顺序，得到一个数据集

我们要如何估计这个数据集中有多少不同的数字呢？因为知道这些数字是服从均匀分布的随机数字，一个比较简单的可行方案是：找出数据集中最小的数字。假如m是数值上限，x是找到的最小的数，则m/x是基数的一个估计。例如，我们扫描一个包含0到1之间数字组成的数据集，其中最小的数是0.01，则一个比较合理的推断是数据集中大约有100个不同的元素，否则我们应该预期能找到一个更小的数。注意这个估计值和重复次数无关：就如最小值重复多少次都不改变最小值的数值。

这个估计方法的优点是十分直观，但是准确度一般。例如，一个只有很少不同数值的数据集却拥有很小的最小值；类似的一个有很多不同值的数据集可能最小值并不小。最后一点，其实只有很少的数据集符合随机均匀分布这一前提。尽管如此，这个原型算法仍然是了解基数估计思想的一个途径；后面我们会了解一些更加精巧的算法。

基数估计的概率算法

最早研究高精度基数估计的论文是Flajolet和Martin的Probabilistic Counting Algorithms for Data Base Applications，后来Flajolet又发表了LogLog counting of large cardinalities和HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm两篇论文对算法进行了进一步改进。通过逐篇阅读这些论文来了解算法的发展和细节固然有趣，不过在这篇文章中我会忽略一些算法的理论细节，把精力主要放在如何通过论文中的算法解决问题。有兴趣的读者可以读一下这三篇论文；本文不会介绍其中的数学细节。

Flajolet和Martin最早发现通过一个良好的哈希函数，可以将任意数据集映射成服从均匀分布的（伪）随机值。根据这一事实，可以将任意数据集变换为均匀分布的随机数集合，然后就可以使用上面的方法进行估计了，不过只是这样是远远不够的。

接下来，他们陆续发现一些其它的基数估计方法，而其中一些方法的效果优于之前提到的方法。Flajolet和Martin计算了哈希值的二进制表示的0前缀，结果发现在随机数集合中，通过计算每一个元素的二进制表示的0前缀，设k为最长的0前缀的长度，则平均来说集合中大约有2^k个不同的元素；我们可以用这个方法估计基数。但是，这仍然不是很理想的估计方法，因为和基于最小值的估计一样，这个方法的方差很大。不过另一方面，这个估计方法比较节省资源：对于32位的哈希值来说，只需要5比特去存储0前缀的长度。

值得一提的是，Flajolet-Martin在最初的论文里通过一种基于bitmap的过程去提高估计算法的准确度。关于这点我就不再详述了，因为这种方法已经被后续论文中更好的方法所取代；对这个细节有兴趣的读者可以去阅读原始论文。

到目前为止，我们这种基于位模式的估计算法给出的结果仍然不够理想。如何进行改进呢？一个直观的改进方法就是使用多个相互独立的哈希函数：通过计算每个哈希函数所产生的最长0前缀，然后取其平均值可以提高算法的精度。

实践表明从统计意义来说这种方法确实可以提高估计的准确度，但是计算哈希值的消耗比较大。另一个更高效的方法就是随机平均（stochastic averaging）。这种方法不是使用多个哈希函数，而是使用一个哈希函数，但是将哈希值的区间按位切分成多个桶（bucket）。例如我们希望取

1024个数进行平均，那么我们可以取哈希值的前10比特作为桶编号，然后计算剩下部分的0前缀长度。这种方法的准确度和多哈希函数方法相当，但是比计算多个哈希效率高很多。

根据上述分析，我们可以给出一个简单的算法实现。这个实现等价于Durand-Flajolet的论文中提出的LogLog算法；不过为了方便，这个实现中统计的是0尾缀而不是0前缀；其效果是等价的。

```
def trailing_zeroes(num):
    """Counts the number of trailing 0 bits in num."""
    if num == 0:
        return 32 # Assumes 32 bit integer inputs!
    p = 0
    while (num >> p) & 1 == 0:
        p += 1
    return p

def estimate_cardinality(values, k):
    """Estimates the number of unique elements in the input set values.

    Arguments:
        values: An iterator of hashable elements to estimate the cardinality of.
        k: The number of bits of hash to use as a bucket number; there will be 2**k buckets.
    """
    num_buckets = 2 ** k
    max_zeroes = [0] * num_buckets
    for value in values:
        h = hash(value)
        bucket = h & (num_buckets - 1) # Mask out the k least significant bits as bucket ID
        bucket_hash = h >> k
        max_zeroes[bucket] = max(max_zeroes[bucket], trailing_zeroes(bucket_hash))
    return 2 ** (float(sum(max_zeroes)) / num_buckets) * num_buckets * 0.79402
```

这段代码实现了我们上面讨论的估计算法：我们计算每个桶的0前缀（或尾缀）的最长长度；然后计算这些长度的平均数；假设平均数是x，桶数量是m，则最终的估计值是$2^x \times m$。其中一个没提过的地方是魔法数字0.79402。统计分析显示这种预测方法存在一个可预测的偏差；这个魔法数字是对这个偏差的修正。实际经验表明计算值随着桶数量的不同而变化，不过当桶数量不太小时（大于64），计算值会收敛于估计值。原论文中描述了这个结论的推导过程。

这个方法给出的估计值比较精确 —— 在分桶数为m的情况下，平均误差为$1.3/\sqrt{m}$。因此对于分桶数为1024的情况（所需内存1024*5 = 5120位，或640字节），大约会有4%的平均误差；每桶5比特的存储已经足以估计2^{27}的数据集，而我们只用的不到1k的内存！

让我们看一下试验结果：

```
>>> [100000/estimate_cardinality([random.random() for i in range(100000)], 10) for j in range(10)]
[0.9825616152548807, 0.9905752876839672, 0.979241749110407, 1.050662616357679, 0.937090578752079, 0.9878968276629505,
0.9812323203117748, 1.0456960262467019, 0.9415413413873975, 0.9608567203911741]
```

不错！虽然有些估计误差大于4%的平均误差，但总体来说效果良好。如果你准备自己做一下这个试验，有一点需要注意：Python内置的 hash() 方法将整数哈希为它自己。因此诸如 estimate_cardinality(range(10000), 10) 这种方式得到的结果不会很理想，因为内置 hash() 对于这种情况并不能生成很好的散列。但是像上面例子中使用随机数会好很多。

提升准确度：SuperLogLog和HyperLogLog

虽然我们已经有了一个不错的估计算法，但是我们还能进一步提升算法的准确度。Durand和Flajolet发现离群点会大大降低估计准确度；如果在计算平均值前丢弃一些特别大的离群值，则可以提高精确度。特别的，通过丢弃最大的30%的桶的值，只使用较小的70%的桶的值来进行平均值计算，则平均误差可以从$1.3/\sqrt{m}$降低到$1.05/\sqrt{m}$！这意味着在我们上面的例子中，使用640个字节可情况下可以将平均误差从4%降低到3.2%，而所需内存并没有增加。

最后，Flajolet等人在HyperLogLog论文中给出一种不同的平均值，使用调和平均数取代几何平均数（译注：原文有误，此处应该是算数平均数）。这一改进可以将平均误差降到$1.04/\sqrt{m}$，而且并没不需要额外资源。但是这个算法比前面的算法复杂很多，因为对于不同基数的数据集要做不同的修正。有兴趣的读者可以阅读原论文。

并行化

这些基数估计算法的一个好处就是非常容易并行化。对于相同分桶数和相同哈希函数的情况，多台机器节点可以独立并行的执行这个算法；最后只要将各个节点计算的同一个桶的最大值做一个简单的合并就可以得到这个桶最终的值。而且这种并行计算的结果和单机计算结果是完全一致的，所需的额外消耗仅仅是小于1k的字节在不同节点间的传输。

结论

基数估计算法使用很少的资源给出数据集基数的一个良好估计，一般只要使用少于1k的空间存储状态。这个方法和数据本身的特征无关，而且可以高效的进行分布式并行计算。估计结果可以用于很多方面，例如流量监控（多少不同IP访问过一个服务器）以及数据库查询优化（例如我们是否需要排序和合并，或者是否需要构建哈希表）。
